Formal Aspects of Computing (1997) 3: 1–000

A Mechanized Refinement Proof for a Garbage Collector

Klaus Havelund*[1] and Natarajan Shankar†[2]

* Dep. of Computer Science,
Aalborg University, Denmark
havelund@cs.auc.dk – `www.cs.auc.dk/~havelund`

† Computer Science Laboratory,
SRI International,
CA 94025 USA
shankar@csl.sri.com – `www.csl.sri.com/~shankar`

Keywords: Garbage Collector; Concurrency; Program Verification; State Transition Systems; Refinement; Theorem Proving; PVS.

Abstract.
We describe how the PVS verification system has been used to verify a safety property of a widely studied garbage collection algorithm. The safety property asserts that "nothing but garbage is ever collected." The garbage collection algorithm and its composition with the user program can be regarded as a concurrent system with two processes working on a shared memory. Such concurrent systems are encoded in PVS as state transition systems using a model similar to TLA [Lam94]. The safety criterion is formulated as a refinement and proved using refinement mappings. Russinoff [Rus94] has verified the algorithm in the Boyer-Moore prover, but his proof is not based on refinement and safety property cannot be appreciated without a glass box view of the workings of the algorithm. Using refinement, however, the safety criterion makes sense independent of the garbage collection algorithm. As a by-product, we encode a a version of the theory of refinement mappings in PVS.

[1] Supported by a European Community HCM grant while working at LITP, Paris 6, France; and by the Danish BRICS project while working at Aalborg University.
[2] Supported by NSF Grant CCR-930044 and by ARPA through NASA Ames Research Center under Contract NASA-NAG-2-891 (ARPA Order A721).
Correspondence and offprint requests to: Klaus Havelund, Aalborg University, Department of Computer Science, Frederik Bajersvej 7E, 9220 Aalborg, Denmark.

1. Introduction

Russinoff [Rus94] used the Boyer-Moore theorem prover to verify a safety property of a *mark–and–sweep* garbage collection algorithm originally suggested by Ben-Ari [BA84]. The garbage collector and its composition with a user program is regarded as a concurrent system with both processes working on a common shared memory. The collector uses a colouring (marking) technique to iteratively colour all accessible nodes *black* while leaving garbage nodes *white*. When the colouring has stabilized, all the white nodes can be collected and placed in the free list.

Russinoff's correctness property is formulated as a state predicate P, which is then proven to be an invariant, i.e., true in all reachable states. In gross terms, this invariant predicate is formulated as follows. The garbage collector can at any time be in one of 9 different locations. In one of the locations, here called APPEND, the *append* operation representing garbage collection is applied to a certain memory node X, but only when this node is white. The safety predicate P is then formulated as:

If the control of the garbage collector is at location APPEND and X is white then X is garbage

This formulation of the safety property does not really tell us whether the program is correct. We have to additionally ensure that the *append* operation is only invoked in location APPEND, and only on white nodes. Hence, the safety property of the garbage collector follows from both the invariance of P and an operational understanding of the garbage collection algorithm.

This observation motivated us to carry out a proof in PVS [ORSvH95] using a refinement approach, where the safety property itself is formulated as an abstract algorithm, and the proof is based on refinement mappings as suggested by Lamport [Lam94]. This approach has the advantage that the safety property can be formulated more abstractly without considering the internal structure of the final implementation. Here a *black box* view of the algorithm is sufficient. This yields a further contribution in terms of the formalization of refinement mappings in PVS. In order better to make a comparison, we also carried out a proof in PVS using the same technique as in [Rus94]. This work is documented in a separate technical report [Hav96].

In [HS96] we verified a distributed communication protocol using similar techniques for representing state transition systems. The work presented here can therefore be seen in the context of a series of PVS verifications of parallel/distributed algorithms, where we try to establish a stable framework for such verifications, while attempting to identify the main difficulties in carrying out such proofs. Our larger goal is to improve the applicability of PVS to problems of this kind. Our key conclusion is that techniques for strengthening invariants are of major importance in refinement proofs as well as in proofs of temporal properties.

An initial version of the algorithm was first proposed by Dijkstra, Lamport, Martin, Scholten, and Steffers [DLM$^+$78] as an exercise in organizing and verifying the cooperation of concurrent processes. Their solution involved three colours. Ben-Ari improved this algorithm so as to use only two colours while simplifying the resulting proof. All of these proofs were informal *pencil and paper* exercises. As pointed out by Russinoff [Rus94], these informal proofs ran into difficulties of one sort or another. Dijkstra, et al [DLM+78] explained (as an example of a "logical trap") how they originally proposed a minor modification to the algorithm. This claim turned out to be wrong, and was discovered by the authors just before the proof reached publication. Ben-Ari later proposed the same modification to his algorithm and argued for its correctness without discovering its flaw. Counterexamples were later given by Pixley [Pix88] and van de Snepscheut [dS87]. Furthermore, although Ben-Ari's algorithm (which is the one we verify in PVS) is correct, his proof of the safety property was found to be flawed. This flaw was essentially reproduced by Pixley[Pix88] where it again survived the review process, and was only discovered ten years later by Russinoff during the course of his mechanical verification [Rus94]. Ben-Ari also gave a flawed proof of a liveness property (*every garbage node will eventually be collected*) that was later observed and corrected by van de Snepscheut [dS87].

The garbage collector has also been specified and verified in the UNITY framework [CM88] using a notion of refinement called *superposition*. This refinement notion differs from ours in the sense that the initial algorithm (specification) is not regarded as a specification of lower levels. Rather it is taken as an initial starting point that is then enriched with more details until the final enrichment represents the full algorithm. Each level inherits the properties proved about the previous level, such that the final level inherits all the properties proven for all previous levels, and this combined collection of properties can then be used to prove that the final level satisfies the desired safety and liveness properties. Only the marking phase is verified in this exercise.

In Section 2, a formalization of state transition systems and refinement mappings is provided in an informal mathematical style that is later formalized in PVS. The garbage collection algorithm is described in Section 3. Sections 4 and 5 present the successive refinements of the initial algorithm in three stages. This presentation is based on an informal notation for transition systems. Sections 6 and 7 formalize the concepts introduced in Sections 2, 4 and 5 in PVS. Finally, Section 8 lists some observations on the entire verification exercise.

2. Transition Systems and Refinement Mappings

In this section, we establish the formal theory for using an abstract non-deterministic program as a safety specification so that any behaviour is safe as long as it is generated by the program. An implementation is then defined as a refinement of this program. The basic concepts are those of *transition systems*, *traces*, *invariants*, *observed transition systems*, *refinements*, and *refinement mappings*. The theory presented is a minor modification of the theory developed by Abadi and Lamport [AL91], with some modest differences. We first introduce the basic concept of a transition system. Specifications as well as their refinements are written as transition systems.

Definition 2.1. (Transition System) A transition system is a triple (Σ, I, N), where

- Σ is a *state* space

- $I \subseteq \Sigma$ is the set of *initial* states.
- $N \subseteq \Sigma \times \Sigma$ is the *next-state* relation. Elements of N are denoted by pairs of the form (s, t), meaning that there is a transition from the state s to the state t.

An *execution trace* is an infinite sequence of states, where the first state satisfies the initiality predicate and every pair of adjacent states is related by the next-state relation. A sequence σ is just an infinite enumeration of states $\langle s_0, s_1, s_2, \ldots \rangle$. We let σ_i denote the i'th element s_i of the sequence. The traces of a transition system can be defined as follows.

Definition 2.2. (Traces) The traces of a transition system are defined as follows:

$$\Theta(\Sigma, I, N) = \{\sigma \in \Sigma^* \mid \sigma_0 \in I \wedge \forall i \geq 0 \cdot N(\sigma_i, \sigma_{i+1})\}$$

We shall need the notion of a transition system invariant, which is a state predicate true in all states reachable from an initial state by following the transition relation.

Definition 2.3. (Invariant) Given a transition system $S = (\Sigma, I, N)$, then a predicate $P : \Sigma \to \mathcal{B}$ is an S invariant iff.

$$\forall \sigma \in \Theta(S) \cdot \forall i \geq 0 \cdot P(\sigma_i)$$

Since we want to compare transition systems, and decide whether one transition system refines another, we need a notion of *observability*. For that purpose, we extend transition systems with an *observation function*, which when applied to a state returns an observation in some domain.

Definition 2.4. (Observed Transition System) An observed transition system is a five-tuple $(\Sigma, \Sigma_o, I, N, \pi)$ where

- (Σ, I, N) is a transition system
- Σ_o is a state space, the observed one
- $\pi : \Sigma \to \Sigma_o$ is an *observation function* that extracts the observed part of a state.

Typically (at least in our case) a state $s \in \Sigma$ consists of an observable part $s_{obs} \in \Sigma_o$ and an internal part s_{int}, hence $s = (s_{obs}, s_{int})$ and π is just the projection function: $\pi(s_{obs}, s_{int}) = s_{obs}$. We adopt the convention that a projection function π applied to a trace $\langle s_1, s_2, \ldots \rangle$ results in the projected trace $\langle \pi(s_1), \pi(s_2), \ldots \rangle$.

The central concept in all this is the notion of refinement: that one observed transition system S_2 refines another observed transition system S_1. By this we intuitively mean that every observation we can make on S_2, we can also make on S_1. Hence, if S_1 behaves safely so will S_2 since every projected trace of S_2 is a projected trace of S_1. This is formulated in the following definition.

Definition 2.5. (Refinement) An observed transition system $S_2 = (\Sigma_2, \Sigma_o, I_2, N_2, \pi_2)$ *refines* an observed transition system $S_1 = (\Sigma_1, \Sigma_o, I_1, N_1, \pi_1)$ iff (note that they have the same observed state space Σ_o):

$$\forall \sigma_2 \in \Theta(S_2) \cdot \exists \sigma_1 \in \Theta(S_1) \cdot \pi_1(\sigma_1) = \pi_2(\sigma_2)$$

We have thus established what it means for one observed transition system to refine another, but we still need a practical way of showing refinement. Note that refinement is defined in terms of traces which are infinite objects so that reasoning about them directly is impractical. We need a way of reasoning about states and pairs of states. A *refinement mapping* is a suitable tool for this purpose. A refinement mapping from a lower level transition system S_2 to a higher-level one S_1 is a mapping from the state space Σ_2 to the state space Σ_1, that when applied statewise, maps traces of S_2 to traces of S_1. This is formally stated as follows.

Definition 2.6. (Refinement Mapping) A *refinement mapping* from an observed transition system $S_2 = (\Sigma_2, \Sigma_o, I_2, N_2, \pi_2)$ to an observed transition system $S_1 = (\Sigma_1, \Sigma_o, I_1, N_1, \pi_1)$ is a mapping $f : \Sigma_2 \rightarrow \Sigma_1$ such that there exists an S_2 invariant P, where:

1. $\forall s \in \Sigma_2 \cdot \pi_1(f(s)) = \pi_2(s)$
2. $\forall s \in \Sigma_2 \cdot I_2(s) \Rightarrow I_1(f(s))$
3. $\forall s, t \in \Sigma_2 \cdot P(s) \wedge P(t) \wedge N_2(s,t) \Rightarrow N_1(f(s), f(t))$

We can now state the main theorem (which is stated in [AL91], and which we have proved in PVS for our slightly modified version):

Theorem 2.1. (Existence of Refinement Mappings) If there exists a refinement mapping from an observed transition system S_2 to an observed transition system S_1, then S_2 refines S_1.

We shall show how we demonstrate the existence of refinement mappings in PVS, by providing a *witness*, that is: defining a particular one. Defining the refinement mapping turns out typically to be easy, whereas showing that it is indeed a refinement mapping (the properties in Definition 2.6) is where the major effort goes. Especially finding and proving the invariant P is the bulk of the proof.

We differ from Abadi and Lamport [AL91] in two ways. First, we allow general observation functions, and not just projection functions that are the identity map on a subset of the state space. Second, in Definition 2.6 of refinement mappings, we assume that states s and t satisfy an implementation invariant P, which is not the case in [AL91]. We have thus weakened the premises of the refinement rule. Whereas the introduction of observation functions is just a nice (but not strictly necessary) generalization, the use of invariants is of real importance for practical proofs.

3. The Algorithm

In this section we informally describe the garbage collection algorithm. As illustrated in Figure 1, the system consists of two processes, the *mutator* and the *collector*, working on a shared *memory*.

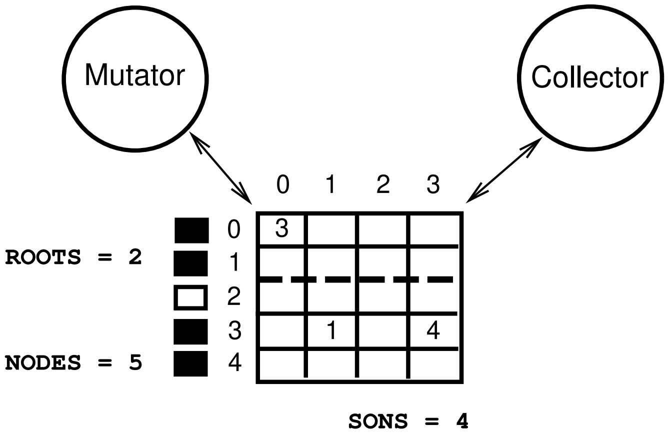

Fig. 1. The Mutator, Collector and Shared Memory

3.1. The Memory

The memory is a fixed size array of *nodes*. In Figure 1 there are 5 nodes (rows) numbered 0–4. Associated with each node is an array of uniform length of *cells*. Figure 1 shows 4 cells numbered 0 – 3 per node. A cell is identified by a pair of integers (n,i) where n is a node number and where i is called the *index*. Each cell contains a pointer to a node, called the *son*. In the case of a LISP implementation, there are, for example, two cells per node. In Figure 1, we assume that all empty cells contain the *NIL* value 0, and hence point to node 0. In addition, node 0 points to node 3 (because cell (0,0) does so), which in turn points to nodes 1 and 4. Hence the memory can be thought of as a two-dimensional array, the size of which is determined by the positive integer constants `NODES` and `SONS`. Each node has an associated *colour*, black or white, that is used by the collector in identifying garbage nodes.

A pre-determined number of nodes, defined by the positive integer constant `ROOTS`, are designated as the *roots*, and these are kept in the initial part of the array (they may be thought of as static program variables). In Figure 1, there are two such roots shown separated from the rest with a dotted line. A node is *accessible* if it can be reached from a root by following pointers, and a node is *garbage* if it is not accessible. Nodes 0, 1, 3, and 4 in Figure 1 are therefore accessible, and node 2 is garbage.

There are only three operations by which the memory structure can be modified:

- Redirect a pointer towards an accessible node.
- Change the colour of a node.
- Append a garbage node to the free list.

In the initial state, all pointers are assumed to be 0, and nothing is assumed about the colours.

3.2. The Mutator

The mutator corresponds to the user program and performs the main computation. From an abstract point of view, it continuously changes pointers in the memory; the changes being arbitrary except for the fact that a cell can only be set to point to an already accessible node. In changing a pointer the "previously pointed-to" node may become garbage, if it is not accessible from the roots in some alternative way. In Figure 1, any cell can hence be modified by the mutator to point to a node other than 2. Only accessible cells can be modified, but as shown below, the algorithm can in fact be proved safe without this restriction. The algorithm is as follows:

1. Select a node n, an index i, and an accessible node k, and assign k to cell (n,i).
2. Colour node k black. Return to step 1.

Each of the two steps is regarded as an atomic instruction.

3.3. The Collector

The collector collects garbage nodes and puts them into a *free list*, from which the mutator may then remove them as they are needed during dynamic storage allocation. Associated with each node is a *colour* field, that is used by the collector during its identification of garbage nodes. Basically, it colours accessible nodes *black*, and at a certain point it collects all *white* nodes, which are then garbage, and puts them into the free list. Figure 1 illustrates the situation at such a point: only node 2 is white since it is the only garbage node. The collector algorithm is as follows:

1. Colour each root black.
2. Examine each pointer in succession. If the source is black and the target is white, colour the target black.
3. Count the black nodes. If the result exceeds the previous count (or if there was no previous count), return to step 2.
4. Examine each node in succession. If a node is white, append it to the free list; if it is black, colour it white. Then return to step 1.

Steps 1–3 constitute the *marking* phase where all accessible nodes are blackened. Each of these steps involves an iteration involving a smaller step that is executed atomically. For example, step 3 consists of several atomic instructions, each counting (or not) a single node.

4. The Specification

We now present the initial specification of the garbage collector. It is presented as a transition system using an informal notation for describing a such. In Section 6 it is described how we encode transition systems in PVS.

We shall assume a data structure representing the memory. The number of nodes in the memory is defined by the constant `NODES`. The type `Node` defines the numbers from 0 to $\texttt{NODES} - 1$. The constant `SONS` defines the number of cells per node. The type `Index` defines the numbers from 0 to $\texttt{SONS} - 1$. Hence, the memory can be thought of a two-dimensional array, and can be declared as follows:[3]

```
var
  M : array[Node,Index] of Node;
```

The memory will be the observed part of the state (Σ_o – see Definition 2.6) throughout all refinements. For example, the node colouring structure and other auxiliary variables that we later add will be internal. Recall that an initial segment of the nodes are roots, the number being defined by the constant `ROOTS`. A number of functions (e.g., for reading the state) and procedures (e.g., for modifying the state) are assumed:

```
function  accessible(n:Node):bool;
function  son(n:Node,i:Index):Node;
procedure set_son(n:Node,i:Index,k:Node);
procedure append_to_free(n:Node);
```

The function `accessible` returns true if its argument node is accessible from one of the roots by following pointers. The function `son` returns the contents of cell `(n,i)`. The procedure `set_son` assigns `k` to the cell identified by `(n,i)`. Hence after the procedure has been called, this cell now points to `k`. The function `append_to_free` appends its argument node to the list of free nodes, assuming that it is a garbage node. The specification consists of the parallel composition of the mutator and the collector. The mutator looks as follows:

```
MODIFY :
  [1] choose n,k:Node; i:Index where accessible(k) ->
        set_son(n,i,k);
        goto MODIFY
      end
```

A program at any time during its execution is in one of a finite collection of locations that are identified by program labels. The above mutator has one such location named `MODIFY`. Associated with each location is a set of numbered (e.g. `[1]`, `[2]`) rules, typically of the form `p -> s`, where `p` is a pre-condition on the state and `s` is an assignment statement. When the program execution is at this location, all rules where the condition `p` is true in the current state are enabled, and a non-deterministic choice is made between them, resulting in the next state being obtained by applying the `s` statement of the chosen rule to the current state. The "`choose x:T where p -> s end`" construct represents a set of such rules, one for each choice of `x` within its type `T`. Hence, the mutator repeatedly chooses two arbitrary nodes `n,k:Node` and an arbitrary index `i:Index` such that `k` is accessible. The cell `(n,i)` is then set to point to `k`. The collector looks as follows:

[3] The actual PVS specification shown on page 19 is actually more abstract and does not specify the memory as being implemented as an array. We use an array implementation here for clarity of presentation.

```
COLLECT :
  [1] choose n:Node where not accessible(n) ->
        append_to_free(n);
        goto COLLECT
      end
```

It repeatedly chooses an arbitrary inaccessible node which is then appended to the free list of nodes. Since the node is not accessible it is a garbage node, hence only garbage nodes are collected (appended), and this is the proper specification of the garbage collector. This yields an abstract specification of the behavior of the collector that is not yet a reasonable implementation. We need to somehow implement the selection of an inaccessible node.

5. The Refinement Steps

In this section we outline how the refinement is carried out in three steps, resulting in the garbage collection algorithm described informally in Section 3. Each refinement is given an individual section, which again is divided into a *program* section presenting the new program, and a *proof* section outlining the refinement proof. According to Theorem 2.1 a refinement can be proved by identifying a refinement mapping from the concrete state space to the abstract state space, see Definition 2.6. Hence, each *proof* section will consist of a definition of such a mapping together with a proof that it is a refinement mapping, focusing on the simulation relation required in item (3) of Definition 2.6. The PVS encoding of the programs is described in Section 6, while the PVS encoding of the refinement proofs is described in Section 7.

5.1. First Refinement : Introducing Colours

5.1.1. The Program

In the first step, the collector is refined to base its search for garbage nodes on a colouring technique. The type `Colour` is defined as `bool`, the set of booleans, assumed to represent the colours *black* (true) and *white* (false). The global state must be extended with a colouring of each node in the memory (not each cell), and a couple of extra auxiliary variables `Q` and `L` used for other purposes. The extended state is shown below.

```
var
  M : array[Node,Index] of Node;
  C : array[Node] of Colour;
  Q : Node;
  L : nat;
```

Three extra operations on this new data structure are needed:

```
procedure set_colour(n:Node,c:Colour);
function  colour(n:Node):Colour;
function  blackened():bool;
```

The procedure `set_colour` colours a node either white or black by updating the concrete variable `C`. The function `colour` returns the colour of a node. Finally, the function `blackened` returns true if *all accessible nodes are black*. The mutator is now refined into the program which was informally described in Section 3:

```
MUTATE :
  [1] choose n,k:Nodes; i:Index where accessible(k) ->
        set_son(n,i,k);
        Q := k;
        goto COLOUR;
      end
COLOUR :
  [1] true -> set_colour(Q,true); goto MUTATE;
```

There are two locations, `MUTATE` and `COLOUR`. In the `MUTATE` location, in addition to the mutation, the target node `k` is assigned to the global auxiliary variable `Q`. Then in the `COLOUR` location, `Q` is coloured black. *Note that the mutator will not be further refined, it will now stay unchanged during the remaining refinements of the collector.* The collector is defined as follows.

```
COLOUR :
  [1] choose n:Nodes ->
        set_colour(n,true);
        goto COLOUR;
      end;
  [2] blackened() -> L := 0; goto TEST_L;
TEST_L :
  [1] L = NODES -> goto COLOUR;
  [2] L < NODES -> goto APPEND;
APPEND :
  [1] not colour(L) -> append_to_free(L); L := L + 1; goto TEST_L;
  [2] colour(L) -> set_colour(L,false); L := L + 1; goto TEST_L;
```

It consists of two phases. While in the `COLOUR` location, nodes are coloured arbitrarily until all accessible nodes are black (`blackened()`). The style in which colouring is expressed may seem surprising, but it is a way of defining a post condition: *colour at least all accessible nodes.*[4] In the second phase at locations `TEST_L` and `APPEND`, all white nodes are regarded as garbage nodes, and are hence collected (appended to the free list). The auxiliary variable `L` is used to control the loop: it runs through all the nodes. After appending all garbage nodes to the free list, the colouring phase is restarted.

5.1.2. The Refinement Proof

The refinement mapping, call it *abs*, from the concrete state space to the abstract state space maps `M` to `M`. Note that such a mapping only needs to be defined for each component of the abstract state, showing how it is generated from components in the concrete state. Hence, the concrete variables `C`, `Q` and `L` are

[4] By formulating this colouring as an iteration, we can avoid introducing a history variable at a lower refinement level. Note that any node can be coloured, not only accessible nodes. This allows a later refinement to colour nodes that originally were accessible, but later have become garbage.

not used for this purpose. This is generally the case for the refinement mappings to follow: they are the identity on the variables occurring in the abstract state. Also program locations have to be mapped. In fact, each program (mutator, collector) can be regarded as having a program counter variable, and we have to show how the abstract program counter is obtained (mapped) from the concrete. Whenever the concrete program is in a particular location l, then the abstract will be in the location $abs(l)$. In the current case, the concrete mutator locations `MUTATE` and `COLOUR` are both mapped to `MODIFY`, while the concrete collector locations `COLOUR`, `TEST_L` and `APPEND` all are mapped to `COLLECT`. This completes the definition of the refinement mapping.

In order to prove Property (3) in Definition 2.6, we associate each transition in the concrete program with a transition in the abstract program, and prove that: "if the concrete transition brings a state s_1 to a state s_2, then the abstract transition brings the state $abs(s_1)$ to the state $abs(s_2)$". We say that the concrete transition, say t_c, *simulates* the abstract transition, say t_a, and write this as $t_c \ll t_a$. Putting all these sub-proofs together will yield a proof of (3). Some of the concrete transitions just simulate a stuttering step (no state change) in the abstract system. This will typically be some of the new transitions associated with *new location names* added to the concrete program. Other concrete transitions have exact counterparts in the abstract program. These are typically transitions associated with *same location names* as in the abstract program. In the following, we will only mention cases that deviate from the above two; i.e., where we add *new location names*, and where the corresponding *new* transitions do *not* simulate a stuttering step in the abstract program.

Hence in our case, `MUTATE.1` $\ll$ `MODIFY.1`, and `APPEND.1` $\ll$ `COLLECT.1` (`APPEND.2` simulates stuttering). In the proof of `APPEND.1` $\ll$ `COLLECT.1`, an invariant is needed about the concrete program:

$$\texttt{collector@APPEND} \wedge \texttt{accessible(L)} \Longrightarrow \texttt{colour(L)}$$

It says that whenever the concrete collector is at the `APPEND` location, and node `L` is accessible, then `L` is also black. From this we can conclude that the `append_to_free` operation is only applied to garbage nodes, since it is only applied to white nodes. Hence, we need to prove an invariant about the concrete program in order to prove the refinement. In general, the proof of these invariants is what really makes the refinement proof non-trivial. To prove the above invariant, we do in fact need to prove a stronger invariant, namely that in locations `TEST_L` and `APPEND`: $\forall n \geq \texttt{L} \cdot \texttt{accessible}(n) \Longrightarrow \texttt{colour}(n)$. This invariant strengthening is typical in our proofs.

5.2. Second Refinement : Colouring by Propagation

5.2.1. The Program

In this step, accessible nodes are coloured through a propagation strategy, where first all roots are coloured, and next all white nodes which have a black father are coloured. The state is extended with an extra auxiliary variable `K` used for controlling the iteration through the roots. The extended state is shown below.

```
var
  M : array[Node,Index] of Node;
  C : array[Node] of Colour;
  Q : Node;
  K, L : nat;
```

Two additional functions are needed:

```
function bw(n:Node,i:Index):bool;
function exists_bw():bool;
```

The function `bw` returns true if `n` is black and `son(n,i)` is white. The function `exists_bw` returns true if there exists a black node, say `n`, that via one of its cells, say `i`, points to a white node. That is: `bw(n,i)`. The collector becomes as follows:

```
COLOUR_ROOTS :
  [1] K = ROOTS -> goto PROPAGATE;
  [2] K < ROOTS -> set_colour(K,true); K := K+1; goto COLOUR_ROOTS;
PROPAGATE :
  [1] choose n:Node; i:Index where bw(n,i) ->
        set_colour(son(n,i),true);
        goto PROPAGATE;
      end;
  [2] not exists_bw() -> L := 0; goto TEST_L;
TEST_L :
  [1] L = NODES -> K := 0; goto COLOUR_ROOTS;
  [2] L < NODES -> goto APPEND;
APPEND :
  [1] not colour(L) -> append_to_free(L); L := L + 1; goto TEST_L;
  [2] colour(L) -> set_colour(L,false); L := L + 1; goto TEST_L;
```

The `COLOUR` location from the previous level has been replaced by the two locations `COLOUR_ROOTS` and `PROPAGATE` (while the append phase is mostly unchanged). In the `COLOUR_ROOTS` location all roots are coloured black, the loop being controlled by the variable `K`. In the `PROPAGATE` location, either there exists no black node with a white son (i.e. `not exists_bw()`), in which case we start collecting (going to location `TEST_L`), or such a node exists, in which case its son is coloured black, and we continue colouring.

5.2.2. The Refinement Proof

The refinement mapping, besides being the identity on identically named entities (variables as well as locations), maps the collector locations `COLOUR_ROOTS` and `PROPAGATE` to `COLOUR`. Hence concrete root colouring as well as concrete propagation are just particular kinds of abstract colourings.

Concerning the transitions, `COLOUR_ROOTS.2` $\ll$ `COLOUR.1`, `PROPAGATE.1` $\ll$ `COLOUR.1`, and `PROPAGATE.2` $\ll$ `COLOUR.2`. In the proof of `PROPAGATE.2` $\ll$ `COLOUR.2`, an invariant is needed about the concrete program:

$$\texttt{collector@PROPAGATE} \Longrightarrow \forall r : \texttt{Root} \cdot \texttt{colour}(r)$$

It states that in location `PROPAGATE` all roots must be coloured. This fact combined with the propagation termination condition `not exists_bw()`: “there does not exist a pointer from a black node to a white node”, will imply the propagation termination condition in `COLOUR.2` of the abstract specification: `blackened()`, which says that “all accessible nodes are coloured”.

5.3. Third Refinement : Propagation by Scans

5.3.1. The Program

In the last refinement, the propagation, represented by the location `PROPAGATE` above, is refined into an algorithm, where all nodes are repeatedly scanned in sequential order, and if black, their sons coloured; until a whole scan does not result in a colouring. The state is extended with auxiliary variables `BC` (*black count*) and `OBC` (*old black count*), used for counting black nodes; and the variables `H`, `I`, and `J` for controlling loops:

```
var
  M   : array[Node,Index] of Node;
  C   : array[Node] of Colour;
  Q   : Node;
  H, I, J, K, BC, OBC : nat;
```

The collector is described below, where transitions have been divided into 4 steps corresponding to the informal description of the algorithm on page 7. Two loops interact (steps 2 and 3). In the first loop, `TEST_I`, `TEST_COLOUR` and `COLOUR_SONS`, all nodes are scanned, and every black node has all its sons coloured. The variables `I` and `J` are used to “walk” through the cells. In the second loop, `TEST_H`, `COUNT` and `COMPARE`, it is counted how many nodes are black. This amount is stored in the variable `BC`, and if this amount exceeds the old black count, stored in the variable `OBC`, then yet another scan is started, and `OBC` is updated. The variable `H` is used to control this loop.

```
- Step 1 : Colour roots
  COLOUR_ROOTS :
    [1] K = ROOTS -> I := 0; goto TEST_I;
    [2] K < ROOTS -> set_colour(K,true); K := K + 1; goto COLOUR_ROOTS;

- Step 2 : Propagate once
  TEST_I :
    [1] I = NODES -> BC := 0; H := 0; goto TEST_H;
    [2] I < NODES -> goto TEST_COLOUR;
  TEST_COLOUR :
    [1] not colour(I) -> I := I + 1; goto TEST_I;
    [2] colour(I)  -> J := 0; goto COLOUR_SONS;
  COLOUR_SONS :
    [1] J = SONS -> I := I + 1; goto TEST_I;
    [2] J < SONS -> set_colour(son(I,J),true); J := J + 1;
        goto COLOUR_SONS;

- Step 3 : Count black nodes
  TEST_H :
    [1] H = NODES -> goto COMPARE;
    [2] H < NODES -> goto COUNT;
  COUNT :
    [1] not colour(H) -> H := H + 1; goto TEST_H;
    [2] colour(H) -> BC := BC + 1; H := H + 1; goto TEST_H;
  COMPARE :
    [1] BC = OBC -> L := 0; goto TEST_L;
    [2] BC /= OBC -> OBC := BD; I := 0; goto TEST_I;

- Step 4 : Append garbage nodes
  TEST_L :
    [1] L = NODES -> BC := 0; OBC := 0; K := 0; goto TEST_I;
    [2] L < NODES -> goto APPEND;
  APPEND :
    [1] not colour(L) -> append_to_free(L); L := L + 1; goto TEST_L;
    [2] colour(L) -> set_colour(L,false); L := L + 1; goto TEST_L;
```

5.3.2. The Refinement Proof

The refinement mapping is the identity, except for six of the locations of the collector. That is, the collector locations `TEST_I`, `TEST_COLOUR`, `COLOUR_SONS`, `TEST_H`, `COUNT`, and `COMPARE` are all mapped to `PROPAGATE`. Concerning the transitions, `COLOUR_SONS.2` $\ll$ `PROPAGATE.1` whereas `COMPARE.1` $\ll$ `PROPAGATE.2`. In the proof of `COLOUR_SONS.2` $\ll$ `PROPAGATE.1`, the following invariant is needed:

$$\texttt{collector@COLOUR\_SONS} \Longrightarrow \texttt{colour(I)}$$

This property implies that the abstract `PROPAGATE.1` transition pre-condition `bw(I,J)` will be true (in case the son is white) or otherwise (if the son is also black), the concrete transition corresponds to a stuttering step (colouring an already black son is the identity function). Correspondingly, in the proof of `COMPARE.1` $\ll$ `PROPAGATE.2`, the following invariant is needed:

$$\texttt{collector@COMPARE} \wedge \texttt{BC = OBC} \Longrightarrow \neg\ \texttt{exists\_bw()}$$

It states that when the collector is in location `COMPARE`, after a counting scan where the number of black nodes have been counted and stored in `BC`, if the number counted equals the previous (old) count `OBC` then there does not exist a pointer from a black node to a white node. Note that `BC = OBC` is the propagation termination condition, and this then corresponds to the termination condition `not exists_bw()` of the abstract transition `PROPAGATE.2`. The proof of these two invariants is quite elaborate, and does in fact compare in size and "look" to the complete proofs in [Hav96] as well as in [Rus94].

6. Formalization in PVS

This section describes how in general transition systems and refinement mappings are encoded in PVS, and in particular how the garbage collector refinement is encoded.

6.1. Transition Systems and their Refinement

Recall from section 2 that an observed transition system is a five-tuple of the form: $(\Sigma, \Sigma_o, I, N, \pi)$ (Definition 2.4). In PVS we model this as a theory with two type definitions, and three function definitions:

```
ots : THEORY
BEGIN
  State   : TYPE = ...
  O_State : TYPE = ...

  proj : [State -> O_State] = ...
  init : [State -> bool] = ...
  next : [State,State -> bool] = ...
END ots
```

The correspondence with the five-tuple is as follows: Σ = `State`, Σ_o = `O_State`, π = `proj`, I = `init` and N = `next`. The `init` function is a predicate on states, while the `next` function is a predicate on pairs of states. We shall formulate the specification of the garbage collector as well as all its refinements in this way. It will become clear below how in particular the function `next` is defined. Now we can define what is a trace (Definition 2.2) and what is an invariant (Definition 2.3). This is done in the theory `Traces`:

```
Traces[State : TYPE] : THEORY
BEGIN
  init : VAR pred[State]
  next : VAR pred[[State,State]]
  sq   : VAR sequence[State]
  n    : VAR nat

  trace(init,next)(sq):bool =
    init(sq(0)) AND
    FORALL n: next(sq(n),sq(n+1))

  p  : VAR pred[State]

  invariant(init,next)(p):bool =
    FORALL (tr:(trace(init,next))): FORALL n: p(tr(n))
END Traces
```

The theory is parameterized with the `State` type of the observed transition system. The `VAR` declarations are just associations of types to names, such that in later definitions, axioms, and lemmas, these names are assumed to have the corresponding types. In addition, axioms and lemmas are assumed to be universally quantified with these names over the types. Note that `pred[T]` in PVS is short for the function space `[T -> bool]`. The type `sequence[T]` is short for `[nat -> T]`; that is: the set of functions from natural numbers to `T`. A sequence of `State`s is hence an infinite enumeration of states. Given a transition system with initiality predicate `init` and next-state relation `next`, a sequence `sq` is a trace of this transition system if `trace(init,next)(sq)` holds. A predicate `p` is an invariant if `invariant(init,next)(p)` holds. That is: if for any trace `tr`, `p` holds in all positions `n` of that trace. Note how the predicate `trace(init,next)` (it is a predicate on sequences) is turned into a type in PVS by surrounding it with parentheses – the type containing all the elements for which the predicate holds, namely all the program traces.

The next notion we introduce in PVS is that of a refinement between two observed transition systems (Definition 2.5). The theory `Refine_Predicate` below defines the function `refines`, which is a predicate on a pair of observed transition systems: a low level **i**mplementation system as the first parameter, and a high level **s**pecification system as as the second parameter. The theory is parameterized with the state space `S_State` of the high level specification theory, the state space `I_State` of the low level implementation theory, and the observed state space `O_State`, which we remember is common for the two observed transition systems. Refinement is defined as follows: *for all traces* `i_tr` *of the implementation system, there exists a trace* `s_tr` *of the specification system, such that when mapping the respective projection functions to the traces, they become equal.* The function map has the type `map : [[D->R] -> [sequence[D] -> sequence[R]]]` and simply applies a function to all the elements of a sequence. Finally, we introduce in the theory `Refinement` the notion of a refinement mapping (Definition 2.6) and its use for proving refinement (Theorem 2.1). The theory is parameterized with a specification *observed transition system* (prefixes `S`), an implementation *observed transition system* (prefixes `I`), an abstraction function `abs`, and an invariant `I_inv` over the implementation system.

```
Refine_Predicate[O_State:TYPE, S_State:TYPE, I_State:TYPE] : THEORY
BEGIN
  IMPORTING Traces
  s_init : VAR pred[S_State]
  s_next : VAR pred[[S_State,S_State]]
  s_proj : VAR [S_State -> O_State]
  i_init : VAR pred[I_State]
  i_next : VAR pred[[I_State,I_State]]
  i_proj : VAR [I_State -> O_State]

  refines(i_init,i_next,i_proj)(s_init,s_next,s_proj):bool =
    FORALL (i_tr:(trace(i_init,i_next))):
      EXISTS (s_tr:(trace(s_init,s_next))):
        map(i_proj,i_tr) = map(s_proj,s_tr)
END Refine_Predicate
```

```
Refinement[
  O_State : TYPE,
  S_State : TYPE,
  S_init  : pred[S_State],
  S_next  : pred[[S_State,S_State]],
  S_proj  : [S_State -> O_State],
  I_State : TYPE,
  I_init  : pred[I_State],
  I_next  : pred[[I_State,I_State]],
  I_proj  : [I_State -> O_State],
  abs     : [I_State -> S_State],
  I_inv   : [I_State -> bool]] : THEORY
BEGIN
  ASSUMING
    IMPORTING Traces
    s     : VAR I_State
    r1,r2 : VAR (I_inv)
    proj_id : ASSUMPTION FORALL s: S_proj(abs(s)) = I_proj(s)
    init_h  : ASSUMPTION FORALL s: I_init(s) IMPLIES S_init(abs(s))
    next_h  : ASSUMPTION I_next(r1,r2) IMPLIES S_next(abs(r1),abs(r2))
    invar   : ASSUMPTION invariant(I_init,I_next)(I_inv)
  ENDASSUMING
  IMPORTING Refine_Predicate[O_State,S_State,I_State]

  ref : THEOREM refines(I_init,I_next,I_proj)(S_init,S_next,S_proj)
END Refinement
```

The theory contains a number of assumptions on the parameters and a theorem, which has been proven using the assumptions. Hence, the way to use this parameterized theory is to apply it to arguments that satisfy the assumptions, prove these, and then obtain as a consequence, the theorem which states that the implementation refines the specification (corresponding to Theorem 2.1). This theorem has been proved once and for all. The assumptions are as stated in Definition 2.6. We shall further need to assume transitivity of the refinement relation, and this is formulated (and proved) in the theory `Refine_Predicate_Transitive`.

```
Refine_Predicate_Transitive[
  O_State:TYPE, State1:TYPE, State2:TYPE, State3:TYPE] : THEORY
BEGIN
  IMPORTING Refine_Predicate
  init1 : VAR pred[State1]
  next1 : VAR pred[[State1,State1]]
  proj1 : VAR [State1 -> O_State]
  init2 : VAR pred[State2]
  next2 : VAR pred[[State2,State2]]
  proj2 : VAR [State2 -> O_State]
  init3 : VAR pred[State3]
  next3 : VAR pred[[State3,State3]]
  proj3 : VAR [State3 -> O_State]

  transitive : LEMMA
    refines[O_State,State2,State3]
      (init3,next3,proj3)(init2,next2,proj2) AND
    refines[O_State,State1,State2]
      (init2,next2,proj2)(init1,next1,proj1)
      IMPLIES
    refines[O_State,State1,State3]
      (init3,next3,proj3)(init1,next1,proj1)
END Refine_Predicate_Transitive
```

6.2. The Specification

In this section we outline how the initial specification from section 4 of the garbage collector is modeled in PVS. We start with the specification of the memory structure, and then continue with the two processes that work on this shared structure.

6.2.1. The Memory

The memory type is introduced in the theory `Memory`, parameterized with the memory boundaries. That is, `NODES`, `SONS`, and `ROOTS` define respectively the number of nodes (rows), the number of sons (columns/cells) per node, and the number of nodes that are roots. They must all be positive natural numbers (different from 0). There is also an obvious assumption that `ROOTS` is not bigger than `NODES`. These three memory boundaries are parameters to all our theories. The `Memory` type is defined as an abstract (non-empty) type upon which a constant and collection of functions are defined. First, however, types of nodes, indexes and roots are defined. The constant `null_array` represents the initial memory containing `0` in all memory cells (axiom `mem_ax1`). The function `son` returns the pointer contained in a particular cell. That is, the expression `son(n,i)(m)` returns the pointer contained in the cell identified by node `n` and index `i`. Finally, the function `set_son` assigns a pointer to a cell. That is, the expression `set_son(n,i,k)(m)` returns the memory `m` updated in cell `(n,i)` to contain (a pointer to node) `k`. In order to define what is an accessible node, we introduce the function `points_to`, which defines what it means for one node, `n1`, to point to another, `n2`, in the memory `m`.

```
Memory[NODES:posnat, SONS:posnat, ROOTS:posnat] : THEORY
BEGIN
  ASSUMING roots_within : ASSUMPTION ROOTS <= NODES ENDASSUMING
  Memory : TYPE+
  Node   : TYPE = {n : nat | n < NODES}
  Index  : TYPE = {i : nat | i < SONS}
  Root   : TYPE = {r : nat | r < ROOTS}
  m         : VAR Memory
  n,n1,n2,k : VAR Node
  i,i1,i2   : VAR Index

  null_array : Memory
  son        : [Node,Index -> [Memory -> Node]]
  set_son    : [Node,Index,Node -> [Memory -> Memory]]

  mem_ax1 : AXIOM son(n,i)(null_array) = 0

  mem_ax2 : AXIOM son(n1,i1)(set_son(n2,i2,k)(m)) =
                  IF n1=n2 AND i1=i2 THEN k ELSE son(n1,i1)(m) ENDIF

  points_to(n1,n2)(m):bool = EXISTS (i:Index): son(n1,i)(m)=n2

  accessible(n)(m): INDUCTIVE bool =
    n < ROOTS OR
    EXISTS k: accessible(k)(m) AND points_to(k,n)(m)

  append_to_free : [Node -> [Memory -> Memory]]
  append_ax: AXIOM (NOT accessible(k)(m)) IMPLIES
                     (accessible(n)(append_to_free(k)(m))
                      IFF (n = k OR accessible(n)(m)))
END Memory
```

The function `accessible` is then defined inductively, yielding the least predicate on nodes n (true on the smallest set of nodes) where either n is a root, or n is pointed to from an already reachable node k. Finally we define the operation for appending a garbage node to the list of free nodes, that can be allocated by the mutator. This operation is defined abstractly, assuming as little as possible about its behaviour. Note that, since the free list is supposed to be part of the memory, we could easily have defined this operation in terms of the functions `son` and `set_son`, but this would have required that we took some design decisions as to how the list was represented (for example where the head of the list should be and whether new elements should be added first or last). The axiom `append_ax` defining the append operation says that *in appending a garbage node, only that node becomes accessible, and the accessibility of all other nodes stays unchanged.*

6.2.2. The Mutator and the Collector

The complete PVS formalization of the garbage collector top level specification presented in section 4 is given below.

```
Garbage_Collector[NODES : posnat, SONS : posnat, ROOTS : posnat] : THEORY
BEGIN
  ASSUMING roots_within : ASSUMPTION ROOTS <= NODES ENDASSUMING
  IMPORTING Memory[NODES,SONS,ROOTS]
  State   : TYPE = Memory
  O_State : TYPE = Memory
  s,s1,s2 : VAR State
  n,k     : VAR Node
  i       : VAR Index

  proj(s):O_State = s

  init(s):bool = (s = null_array)

  Rule_mutate(n,i,k)(s):State =
    IF accessible(k)(s) THEN
      set_son(n,i,k)(s)
    ELSE s ENDIF

  Rule_append(n)(s):State =
    IF NOT accessible(n)(s) THEN
      append_to_free(n)(s)
    ELSE s ENDIF

  next(s1,s2):bool =
    (EXISTS n,i,k: s2 = Rule_mutate(n,i,k)(s1)) OR
    (EXISTS n: s2 = Rule_append(n)(s1)) OR
    s2 = s1
END Garbage_Collector
```

The state is simply the memory, and so is the observable state. Hence, there are no hidden variables, and the projection function `proj` is the identity. The next-state relation `next` is defined as a disjunction between three disjuncts, each representing a possible single transition of the total system. The first two disjuncts represent a move of the mutator and the collector, respectively, each move defined through a function. The third possibility just represents *stuttering*: the fact that a process does not change the state (needed for technical reasons).

Since each process (mutator, collector) only has one location we do not model these locations explicitly. The function `Rule_mutate` represents a move by the mutator, which is non-deterministic in the choice of the nodes `n,k` and index `i`. The function, when applied to an *old* state, yields a *new* state, where (if `k` is accessible) a pointer has been changed. Non-deterministic choices are modeled via existential quantifications. Each transition function is defined in terms of an `IF-THEN-ELSE` expression, where the condition represents the guard of the transition (the situation where the transition may meaningfully be applied), and where the `ELSE` part returns the unchanged state, in case the guard is false[5]. The function `Rule_append` represents a move by the collector. In each step, either the

[5] This allows for *stuttering* where rules are applied without changing the state.

mutator makes a move, or the collector does. This corresponds to an interleaving semantics of concurrency. Note how the repeated execution is guaranteed by our interpretation of what is a trace in terms of the next-state relation.

6.3. The First Refinement

In this section we outline how the first refinement from section 5.1 of the garbage collector is modeled in PVS. In order to keep the presentation reasonably sized, we only illustrate this first refinement. The remaining refinements follow the same pattern. First, we describe a collection of colouring functions. The theory `Coloured_Memory` below introduces the primitives needed for colouring memory nodes. The type `Colour` represents the colours *black* (true) and *white* (false). The type `Colours` contains possible colourings of the memory, each being a mapping from nodes to their colours. The functions `colour`, `set_colour` and `blackened` are formalizations of those presented on page 9.

```
Coloured_Memory[NODES : posnat, SONS : posnat, ROOTS : posnat] : THEORY
BEGIN
  ASSUMING roots_within : ASSUMPTION ROOTS <= NODES ENDASSUMING
  IMPORTING Memory[NODES,SONS,ROOTS]
  Colour  : TYPE = bool
  Colours : TYPE = [Node -> Colour]
  n  : VAR Node
  i  : VAR Index
  c  : VAR Colour
  cs : VAR Colours
  m  : VAR Memory

  colour(n)(cs):Colour = cs(n)

  set_colour(n,c)(cs):Colours = cs WITH [n := c]

  blackened(cs,m):bool = FORALL n: accessible(n)(m) IMPLIES colour(n)(cs)
END Coloured_Memory
```

We now show how the first refinement is formulated in PVS. The entire theory called `Garbage_Collector1` is presented in the figure below. First of all, the state type is a record type with a field for each program variable. In addition to the ordinary program variables, there is a program counter “variable” for each process: `MU` for the mutator, and `CHI` for the collector. Each program counter ranges over a type that contains the possible labels. The observed state is still just the memory, hence ignoring, for example, the colouring `C`. We see that the mutator next-state relation `MUTATOR` is now defined as a disjunction between a *mutate* transition and a *colour* Q transition. The collector next-state relation `COLLECTOR` is defined as the disjunction between six possible transitions.

```
Garbage_Collector1[NODES:posnat, SONS:posnat, ROOTS:posnat] : THEORY
BEGIN
  ASSUMING roots_within : ASSUMPTION ROOTS <= NODES ENDASSUMING
  IMPORTING Coloured_Memory[NODES,SONS,ROOTS]
  MuPC    : TYPE = {MUTATE,COLOUR}
  CoPC    : TYPE = {COLOUR,TEST_L,APPEND}
  State   : TYPE = [# MU : MuPC, CHI : CoPC,
                      Q : nat, L : nat , C : Colours, M : Memory #]
  O_State : TYPE = Memory
  s,s1,s2 : VAR State
  n,k     : VAR Node
  i       : VAR Index

  proj(s):O_State = M(s)

  init(s):bool = MU(s) = MUTATE & CHI(s) = COLOUR & M(s) = null_array

  Rule_mutate(n,i,k)(s):State =
    IF MU(s) = MUTATE AND accessible(k)(M(s)) THEN
      s WITH [M := set_son(n,i,k)(M(s)), Q := k, MU := COLOUR]
    ELSE s ENDIF

  Rule_colour_target(s):State =
    IF MU(s) = COLOUR AND Q(s) < NODES THEN
      s WITH [C := set_colour(Q(s),TRUE)(C(s)), MU := MUTATE]
    ELSE s ENDIF

  MUTATOR(s1,s2):bool =
    (EXISTS n,i,k: s2 = Rule_mutate(n,i,k)(s1)) OR
    s2 = Rule_colour_target(s1)

  Rule_stop_colouring(s):State =
    IF CHI(s) = COLOUR AND blackened(C(s),M(s)) THEN
      s WITH [L := 0, CHI := TEST_L]
    ELSE s ENDIF

  Rule_colour(n)(s):State =
    IF CHI(s) = COLOUR THEN
      s WITH [C := set_colour(n,TRUE)(C(s))]
    ELSE s ENDIF

  Rule_stop_appending(s):State =
    IF CHI(s) = TEST_L AND L(s) = NODES THEN
      s WITH [CHI := COLOUR]
    ELSE s ENDIF

  Rule_continue_appending(s):State =
    IF CHI(s) = TEST_L AND L(s) < NODES THEN
      s WITH [CHI := APPEND]
    ELSE s ENDIF

  Rule_black_to_white(s):State =
    IF CHI(s) = APPEND AND L(s) < NODES AND colour(L(s))(C(s)) THEN
      s WITH [C:=set_colour(L(s),FALSE)(C(s)),L:=L(s)+1,CHI:=TEST_L]
    ELSE s ENDIF

  Rule_append_white(s):State =
    IF CHI(s) = APPEND AND L(s) < NODES AND NOT colour(L(s))(C(s)) THEN
      s WITH [M := append_to_free(L(s))(M(s)),L:=L(s)+1,CHI:=TEST_L]
    ELSE s ENDIF

  COLLECTOR(s1,s2):bool =
       s2 = Rule_stop_colouring(s1) OR (EXISTS n:s2 = Rule_colour(n)(s1))
    OR s2 = Rule_stop_appending(s1) OR s2 = Rule_continue_appending(s1)
    OR s2 = Rule_black_to_white(s1) OR s2 = Rule_append_white(s1)

  next(s1,s2):bool = MUTATOR(s1,s2) OR COLLECTOR(s1,s2) OR s2 = s1
END Garbage_Collector1
```

7. The Proof in PVS

The proof of a single refinement lemma (step) is divided into three activities: discovery and proof of *function lemmas*; discovery and proof of *invariant lemmas*; and proof of the *refinement lemma*. A *function lemma* states a property of one or more auxiliary functions involved, which in our case are for example properties about the functions `accessible` and `blackened`. An invariant is a predicate on states, and an *invariant lemma* states that an invariant holds in every reachable state of the concrete implementation (`Garbage_Collector1` in our case). Recall that we needed such an invariant when applying the `Refinement` theory (page 17). The function lemmas are used in proofs of invariant lemmas, which again are used in proofs of *refinement lemmas*.

We shall show these lemmas for the first refinement, using a bottom-up presentation for pedagogical reasons, starting with function lemmas, and ending with the refinement lemma. In, reality, however, the proof was "discovered" top down: the refinement lemma was stated (by applying the `Refinement` theory to proper arguments), and during the proof of the corresponding `ASSUMPTION`s, the need for invariant lemmas were discovered, and during their proofs, function lemmas were discovered.

7.1. Function Lemmas

During the proof, we need a new set of auxiliary functions to "observe" (or calculate) certain values based on the current state of the memory. These *observer functions* occur in invariants. In the first refinement step, we shall need the function `blackened` defined in the theory `Memory_Observers`

```
Memory_Observers[NODES:posnat, SONS:posnat, ROOTS:posnat] : THEORY
BEGIN
  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING
  IMPORTING Coloured_Memory[NODES,SONS,ROOTS]
  cs : VAR Colours
  m  : VAR Memory
  n  : VAR Node
  N  : VAR nat

  blackened(N)(cs,m):bool =
    FORALL (n | N <= n): accessible(n)(m) IMPLIES colour(n)(cs)
  ...
END Memory_Observers
```

This function is similar to the function which is part of the first refinement, page 21, except that it has an additional natural number argument. The function returns true if all nodes above (and including) its argument are black if accessible. The theory contains other functions, but these are first needed in later refinements and will not be discussed here. The lemmas about auxiliary functions that we need for the first refinement are given in the theory `Memory_Properties` below.

```
Memory_Properties[NODES:posnat, SONS:posnat, ROOTS:posnat] : THEORY
BEGIN
  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING
  IMPORTING Memory_Observers[NODES,SONS,ROOTS]

  cs        : VAR Colours
  c         : VAR Colour
  m         : VAR Memory
  n,n1,n2,k : VAR Node
  i,i1,i2,j : VAR Index
  N,N1,N2   : VAR nat

  accessible1 : LEMMA
    accessible(k)(m) AND accessible(n1)(set_son(n,i,k)(m))
      IMPLIES accessible(n1)(m)

  blackened1 : LEMMA
    blackened(n)(cs,m) AND accessible(n)(m) IMPLIES colour(n)(cs)

  blackened2 : LEMMA
    accessible(k)(m) AND blackened(N)(cs,m)
      IMPLIES blackened(N)(cs,set_son(n,i,k)(m))

  blackened3 : LEMMA
    blackened(N)(cs,m) IMPLIES blackened(N)(set_colour(n,TRUE)(cs),m)

  blackened4 : LEMMA
    blackened(n)(cs,m) IMPLIES blackened(n+1)(set_colour(n,FALSE)(cs),m)

  blackened5 : LEMMA
    NOT accessible(n)(m) AND blackened(n)(cs,m)
      IMPLIES blackened(n+1)(cs,append_to_free(n)(m))

  blackened6 : LEMMA
    blackened(cs,m) IMPLIES blackened(0)(cs,m)
END Memory_Properties
```

The theory in its entirety contains other lemmas, needed for later refinements, which we shall however not present here. The lemma `accessible1` is a key lemma, and it says that the `set_son` operator cannot turn garbage nodes into accessible nodes.

7.2. Invariant Lemmas

We can now state the invariant needed for the first refinement step. This is given in the theory `Garbage_Collector1_Inv`. The invariant really needed for the refinement proof is `inv1`, corresponding to the invariant on page 11; but during the proof of that, invariant `inv2` is needed.

```
Garbage_Collector1_Inv[NODES:posnat, SONS:posnat, ROOTS:posnat] : THEORY
BEGIN
  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING
  IMPORTING Memory_Properties[NODES,SONS,ROOTS]
  IMPORTING Garbage_Collector1[NODES,SONS,ROOTS]
  IMPORTING Invariant_Predicates[State]
  s : VAR State

  inv1(s):bool =
    CHI(s)=APPEND AND L(s) < NODES AND accessible(L(s))(M(s))
      IMPLIES colour(L(s))(C(s))

  inv2(s):bool =
    CHI(s)=TEST_L OR CHI(s)=APPEND IMPLIES blackened(L(s))(C(s),M(s))

  I : pred[State] = inv1 & inv2

  inv : LEMMA invariant(init,next)(I)
END Garbage_Collector1_Inv
```

Invariant `inv1` is in fact the safety property originally formulated for the garbage collector [Rus94]. Its proof requires a generalization, which is `inv2`. This shows an example, where we have to strengthen an invariant (`inv1`) to a stronger invariant (`inv2`), which is then proven instead.

7.3. The Refinement Lemma

The first refinement step is formulated as an application of the `Refinement` theory which we defined on page 17. This is done in the theory `Refinement1` shown below.

```
Refinement1[NODES:posnat, SONS:posnat, ROOTS:posnat] : THEORY
BEGIN
  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING
  S  : THEORY = Garbage_Collector [NODES,SONS,ROOTS]
  I1 : THEORY = Garbage_Collector1[NODES,SONS,ROOTS]
  IMPORTING Garbage_Collector1_Inv[NODES,SONS,ROOTS]
  s     : VAR I1.State
  r1,r2 : VAR (I)
  n,k   : VAR Node
  i     : VAR Index
  cs    : VAR Colours

  abs(s):S.State = M(s)

  ...

  R1 : THEORY =
    Refinement[S.O_State,
      S.State,S.init,S.next,S.proj,
      I1.State,I1.init,I1.next,I1.proj,
      abs,I]
END Refinement1
```

The theory imports the specification garbage collector Garbage_Collector, giving it the name S; the implementation Garbage_Collector1, named I1; and the implementation invariant I defined in the theory Garbage_Collector1_Inv. The theory further defines the abstraction function abs, and finally applies the Refinement theory. This application gives rise to four TCCs (Type Checking Conditions) generated by PVS, which have to be proven in order for the PVS specification to be well formed (type check). Furthermore, the proof of these TCCs yields the correctness of the refinement. The TCCs are shown below:

```
R1_TCC1: OBLIGATION FORALL s: S.proj(abs(s)) = I1.proj(s);

R1_TCC2: OBLIGATION FORALL s: I1.init(s) IMPLIES S.init(abs(s));

R1_TCC3: OBLIGATION (FORALL (r1: (I), r2: (I)):
                      I1.next(r1, r2) IMPLIES S.next(abs(r1), abs(r2)));

R1_TCC4: OBLIGATION invariant(I1.init, I1.next)(I);
```

There is a TCC for each ASSUMPTION of the Refinement theory. In particular R1_TCC3 states the simulation property, and R1_TCC4 states the invariant property. As illustrated in section 5.1.2 page 11, we show for each concrete transition which abstract transition it simulates, for example we had that APPEND.1 $\ll$ COLLECT.1, which in this PVS setting is formulated as the following lemma:

```
  sim_append_white : LEMMA
    r2 = Rule_append_white(r1) IMPLIES
      (EXISTS n: abs(r2) = Rule_append(n)(abs(r1))) OR abs(r2) = abs(r1)
```

The technique illustrated above for the first refinement step is repeated for the next two, yielding two further theories Refinement2 and Refinement3. All 3 refinements can now be composed, and the bottom level implementation can be shown to refine the top level specification using transitivity of the refinement relation. This is expressed in the theory Composed_Refinement below, where the theorem ref is our main correctness criteria.

```
Composed_Refinement[NODES:posnat, SONS:posnat, ROOTS:posnat] : THEORY
BEGIN
  ASSUMING
    roots_within : ASSUMPTION ROOTS <= NODES
  ENDASSUMING
  IMPORTING Refinement1[NODES,SONS,ROOTS]
  IMPORTING Refinement2[NODES,SONS,ROOTS]
  IMPORTING Refinement3[NODES,SONS,ROOTS]
  IMPORTING Refine_Predicate
  IMPORTING Refine_Predicate_Transitive

  ref2 : LEMMA
    refines[S.O_State,S.State,I2.State]
      (I2.init,I2.next,I2.proj)(S.init,S.next,S.proj)

  ref : THEOREM
    refines[S.O_State,S.State,I3.State]
      (I3.init,I3.next,I3.proj)(S.init,S.next,S.proj)
END Composed_Refinement
```

8. Observations

It is possible to compare the present proof (PVS_{ref}-proof) with two other mechanized proofs of exactly the same algorithm: the proof in the Boyer-Moore prover [Rus94], from now on referred to as the BM_{inv}-proof; and the PVS proof [Hav96], referred to as the PVS_{inv}-proof. Instead of being based on refinement, these two proofs are based on a statement of the correctness criteria as an invariant to be proven about the implementation (the third refinement step). The PVS_{inv}-proof follows the BM_{inv}-proof closely. Basically the same invariants were used. The PVS_{ref}-proof has the advantage over the two other proofs, that the correctness criteria can be appreciated without knowing the internal structure of the implementation. That is, we do not need to know for example that the append operation is only applied in location APPEND to node X, and only if X is white. Hence, from this perspective, the refinement proof represents an improvement. The PVS_{ref}-proof has approximately the same size as the PVS_{inv}-proof, in that basically the same invariants and lemmas about auxiliary functions need to be proven (19 invariant lemmas and 57 function lemmas). The proof effort took a couple of months. Hence, one cannot argue that the proof has become any simpler. On the contrary in fact: since we have many levels, there is more to prove. Some invariants were easier to discover when using refinement, especially at the top levels. In particular nested loops may be treated nicely with refinement, only introducing one loop at a time. In general, loops in the algorithm to be verified are the reason why invariant discovery is hard, and of course nested loops are no better. The main lesson obtained from the PVS_{inv}-proof is the importance of invariant discovery in safety proofs. Our experience with the PVS_{ref}-proof is that refinement does not relieve us of the need to search for invariants. We had to come up with exactly the same invariants in both cases, but the discovery process was different, and perhaps more structured in the refinement proof.

For future work one can consider investigating ways of automatically generating invariants. Since this is very hard in general, one can alternatively consider more pragmatics ways of supporting the PVS user in identifying new invariant lemmas. For example, a tool that helps the user to infer new invariants from unproven sequents generated by the PVS system as a result of failed proofs.

References

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[BA84] M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Toplas*, 6, July 1984.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.

[DLM+78] E. W. Dijkstra, L. Lamport, A.J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *ACM*, 21, November 1978.

[dS87] J. L. A. Van de Snepscheut. "algorithms for on-the-fly garbage collection" revisited. *Information Processing Letters*, 24, March 1987.

[Hav96] K. Havelund. Mechanical verification of a garbage collector. May 1996.

[HS96] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In M-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681. Springer-Verlag, 1996.

[Lam94] L. Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, 1994.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Pix88] C. Pixley. An incremental garbage collection algorithm for multi-mutator systems. *Distributed Computing*, 3, 1988.

[Rus94] David M. Russinoff. A mechanically verified incremental garbage collector. *Formal Aspects of Computing*, 6:359–390, 1994.